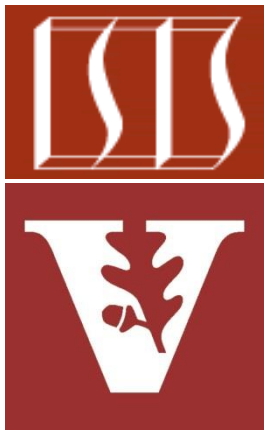
Implementing & Applying Java Atomic Operations



Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Understand how Java atomic classes & operations provide concurrent programs with lock-free, thread-safe mechanisms to read from & write to single variables
- Note a human known use of atomic operations
- Know how Java atomic operations are implemented & applied
  - i.e., implemented in the Java execution environment (e.g., JVM, ART, etc.) level & applied in the Java class library

Concurrency

And few words about concurrency with `Unsafe`. `compareAndSwap` methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```
interface Counter {
    void increment();
    long getCounter();
}
```

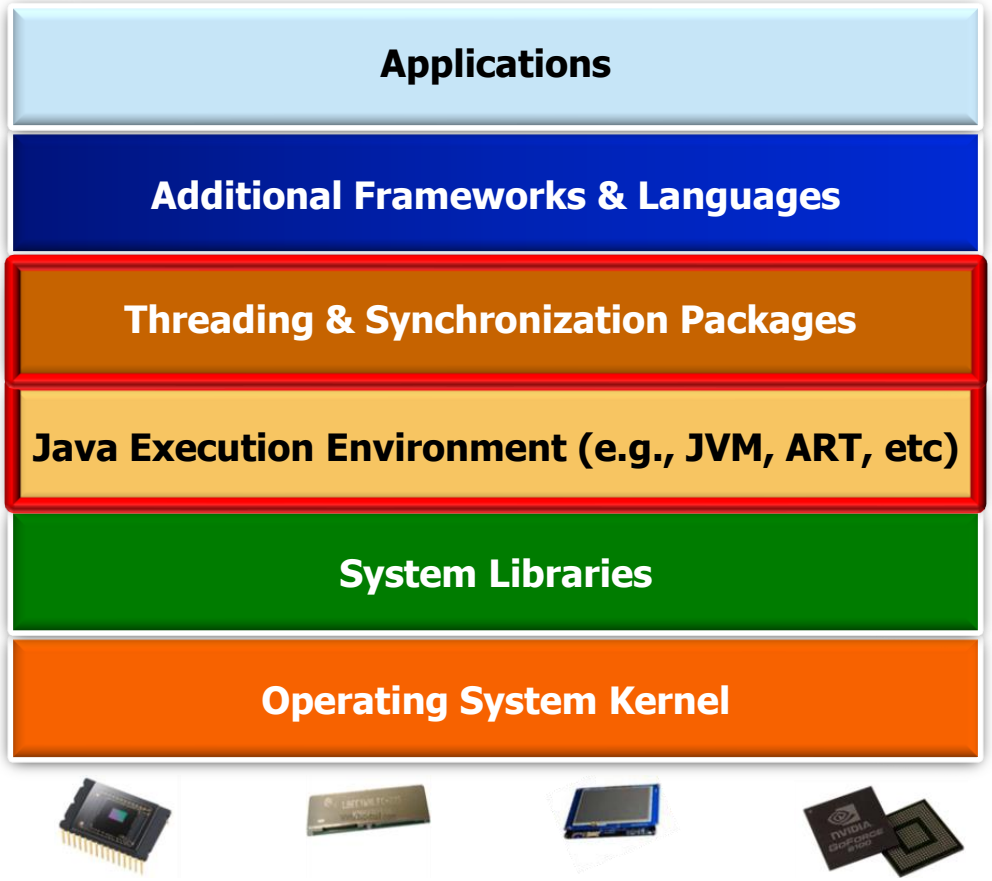
Then we define worker thread `CounterClient`, that uses `Counter`:

```
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```

Implementing Java Atomic Operations

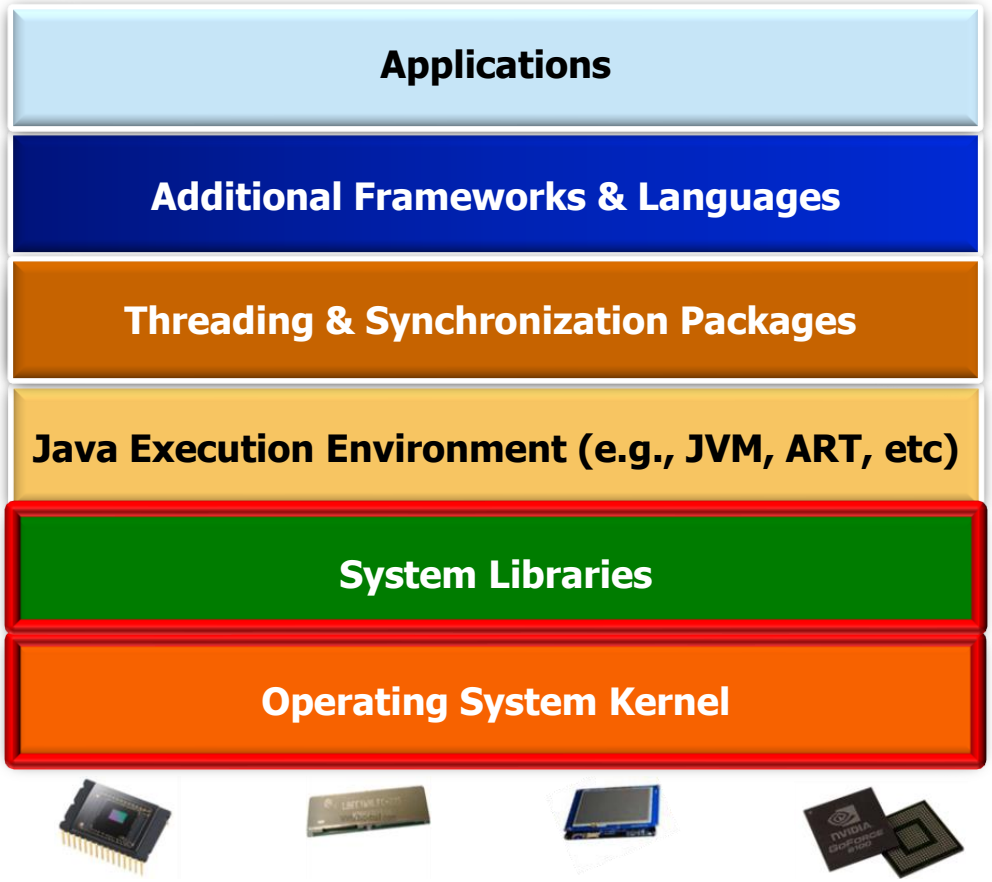
Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of the packages java.util.concurrent*



Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of the packages java.util.concurrent*
  - CAS implementations reside in lower-layers of the stack



Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
  public final native boolean
    compareAndSwapLong(Object o,
                       long offset,
                       long expected,
                       long updated);
```

See www.docjar.com/html/api/sun/misc/Unsafe.java.html

Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
  public final native boolean
    compareAndSwapLong(Object o,
                       long offset,
                       long expected,
                       long updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected)
      base[offset] = updated;
    END_ATOMIC();
    return oldValue;
  }
  ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value

See www.docjar.com/html/api/sun/misc/Unsafe.java.html

Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
  public final native boolean
    compareAndSwapLong(Object o,
                       long offset,
                       long expected,
                       long updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected)
      base[offset] = updated;
    END_ATOMIC();
    return oldValue;
  }
  ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value

Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
  public final native boolean
    compareAndSwapLong(Object o,
                       long offset,
                       long expected,
                       long updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected)
      base[offset] = updated;
    END_ATOMIC();
    return oldValue;
  }
  ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value

Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
  public final native boolean
    compareAndSwapLong(Object o,
                       long offset,
                       long expected,
                       long updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected)
      base[offset] = updated;
    END_ATOMIC();
    return oldValue;
  }
  ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value

Implementing Java Atomic Operations

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*
  - e.g., compareAndSwapLong()

```
public final class Unsafe {
  public final native boolean
    compareAndSwapLong(Object o,
                       long offset,
                       long expected,
                       long updated) {
    START_ATOMIC();
    int *base = (int *) o;
    int oldValue = base[offset];
    if (oldValue == expected)
      base[offset] = updated;
    END_ATOMIC();
    return oldValue;
  }
  ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value

Applying Java Atomic Operations

Applying Java Atomic Operations

- The low-level CAS operations in the Java Unsafe class can only be used in limited circumstances


```
public class AtomicBoolean ... {
  ...
  private static final long
    valueOffset;
  private volatile int value;

  static { ...
    valueOffset = unsafe
      .objectFieldOffset
        (AtomicBoolean.class.
          getDeclaredField("value"));
  }

  public final boolean compareAndSet
    (boolean expected, boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
        (this, valueOffset, e, u);
  } ...
```

Applying Java Atomic Operations

- The low-level CAS operations in the Java Unsafe class can only be used in limited circumstances
  - e.g., within the Java Class Library itself in Java 8 & before

```
public class AtomicBoolean ... {
  ...
  private static final long
    valueOffset;
  private volatile int value;

  static { ...
    valueOffset = unsafe
      .objectFieldOffset
        (AtomicBoolean.class.
          getDeclaredField("value"));
  }

  public final boolean compareAndSet
    (boolean expected, boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
        (this, valueOffset, e, u);
  } ...
```

See classes/java/util/concurrent/atomic/AtomicBoolean.java

Applying Java Atomic Operations

- The low-level CAS operations in the Java Unsafe class can only be used in limited circumstances
  - e.g., within the Java Class Library itself in Java 8 & before

```
public class AtomicBoolean ... {
  ...
  private static final long
    valueOffset;
  private volatile int value;

  static { ...
    valueOffset = unsafe
      .objectFieldOffset
        (AtomicBoolean.class.
          getDeclaredField("value"));
  }

  public final boolean compareAndSet
    (boolean expected, boolean updated){
    int e = expected ? 1 : 0;
    int u = updated ? 1 : 0;
    return unsafe.compareAndSwapInt
        (this, valueOffset, e, u);
  } ...
```

See upcoming lesson on "*Implementing Java AtomicBoolean*" for details

Applying Java Atomic Operations

- Therefore, Java 9+ added a new class called VarHandle that is also designed to be usable by apps, unlike the Java Unsafe class

Class VarHandle

java.lang.Object
    java.lang.invoke.VarHandle

```
public abstract class VarHandle
extends Object
```

A VarHandle is a dynamically strongly typed reference to a variable, or to a parametrically-defined family of variables, including static fields, non-static fields, array elements, or components of an off-heap data structure. Access to such variables is supported under various *access modes*, including plain read/write access, volatile read/write access, and compare-and-swap.

VarHandles are immutable and have no visible state. VarHandles cannot be subclassed by the user.

See docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html

Applying Java Atomic Operations

- Therefore, Java 9+ added a new class called VarHandle that is also designed to be usable by apps, unlike the Java Unsafe class
  - A VarHandle is mostly used for atomic or ordered operations
    - e.g., CAS operations, atomic increment/decrement of fields, etc.

```
class AtomicBoolean ... {
  private static final VarHandle
    VALUE;
  static {
    try {
      VALUE = l.findVarHandle
        (AtomicBoolean.class,
         "value", int.class);
    } ...
  private volatile int value;
  ...
  public final boolean compareAndSet
    (boolean expected,
     boolean updated) {
       return VALUE
        .compareAndSet(this,
           (expected ? 1 : 0),
           (updated ? 1 : 0));
  }
  ...
```

See classes/java/util/concurrent/atomic/AtomicBoolean.java

Applying Java Atomic Operations

- Therefore, Java 9+ added a new class called VarHandle that is also designed to be usable by apps, unlike the Java Unsafe class
  - A VarHandle is mostly used for atomic or ordered operations
    - e.g., CAS operations, atomic increment/decrement of fields, etc.

```
class AtomicBoolean ... {
  private static final VarHandle
    VALUE;
  static {
    try {
      VALUE = l.findVarHandle
        (AtomicBoolean.class,
         "value", int.class);
    } ...
  private volatile int value;
  ...
  public final boolean compareAndSet
    (boolean expected,
     boolean updated) {
       return VALUE
        .compareAndSet(this,
           (expected ? 1 : 0),
           (updated ? 1 : 0));
  }
  ...
```

See upcoming lesson on "*Implementing Java AtomicBoolean*" for details

End of Implementing & Applying Java Atomic Operations